OOSD Assignment 2

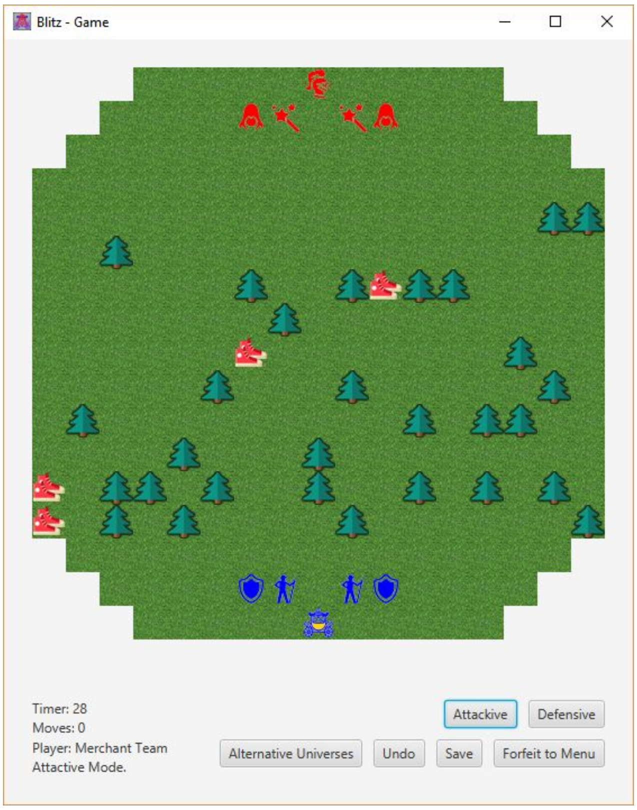
Liam Scanlon 3727321
Shajie Chen 3582098
Sheng Kai Chen 3523185
Priyanga Dhilip Kumar 3670781

Game Features

- Undo 1-3 moves, once per team.
- Additional combat capability (attack / defence).
    - Defense gives 1 extra power, but halves movement.
- Pickup shoes to add 1 movement.
- Save and reopen game at any time.
- 30 second move timer.
- Obstacles introduced on game board.
- Alternative universes, can rewind to any previous tree history node.

Aim of the game is to get the gold merchant from the bottom of the map to the top of the map, escaping past the bandit team. The bandits' aim is to stop the merchant before he reaches the top.

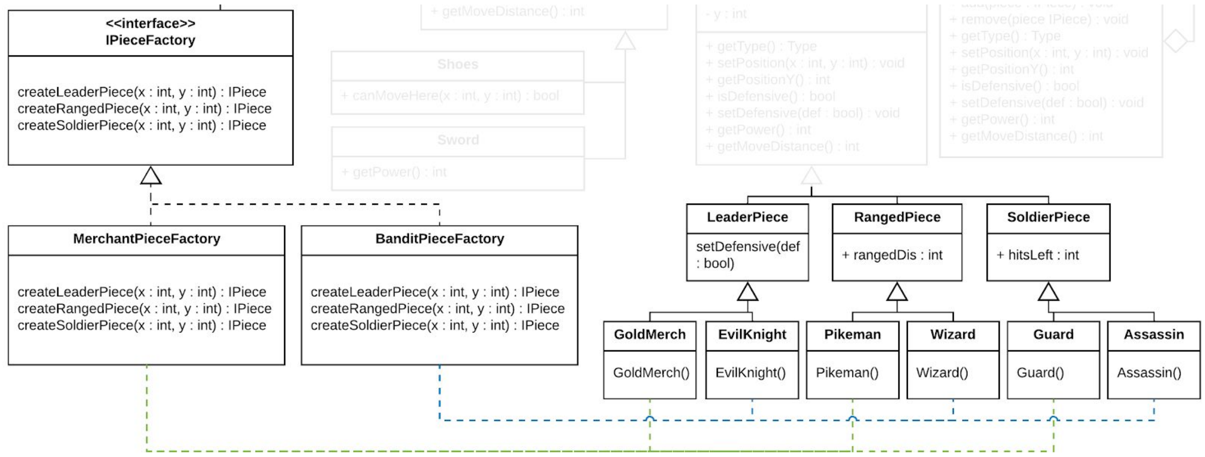
Click help from the main menu to view default power and move capabilities.



Timer: 28
Moves: 0
Player: Merchant Team
Attactive Mode.

Creational Category:

Abstract Factory Pattern:



<<interface>>
IPieceFactory
createLeaderPiece(x : int, y : int) : IPiece
createRangedPiece(x : int, y : int) : IPiece
createSoldierPiece(x : int, y : int) : IPiece
MerchantPieceFactory
createLeaderPiece(x : int, y : int) : IPiece
createRangedPiece(x : int, y : int) : IPiece
createSoldierPiece(x : int, y : int) : IPiece
BanditPieceFactory
createLeaderPiece(x : int, y : int) : IPiece
createRangedPiece(x : int, y : int) : IPiece
createSoldierPiece(x : int, y : int) : IPiece
+ getMoveDistance() : int
Shoes
+ canMoveHere(x : int, y : int) : bool
Sword
+ getPower() : int
- y : int
+ getType() : Type
+ setPosition(x : int, y : int) : void
+ getPositionY() : int
+ isDefensive() : bool
+ setDefensive(def : bool) : void
+ getPower() : int
+ getMoveDistance() : int
+ remove(piece IPiece) : void
+ getType() : Type
+ setPosition(x : int, y : int) : void
+ getPositionY() : int
+ isDefensive() : bool
+ setDefensive(def : bool) : void
+ getPower() : int
+ getMoveDistance() : int
LeaderPiece
setDefensive(def : bool)
RangedPiece
+ rangedDis : int
SoldierPiece
+ hitsLeft : int
GoldMerch
GoldMerch()
EvilKnight
EvilKnight()
Pikeman
Pikeman()
Wizard
Wizard()
Guard
Guard()
Assassin
Assassin()

[Motivation]:

Typically said, the family of related piece object is designed to be used together, and the game needs to enforce this constraint. And also using abstract factory could make the generator independent of how the pieces are created, composed, and represented.

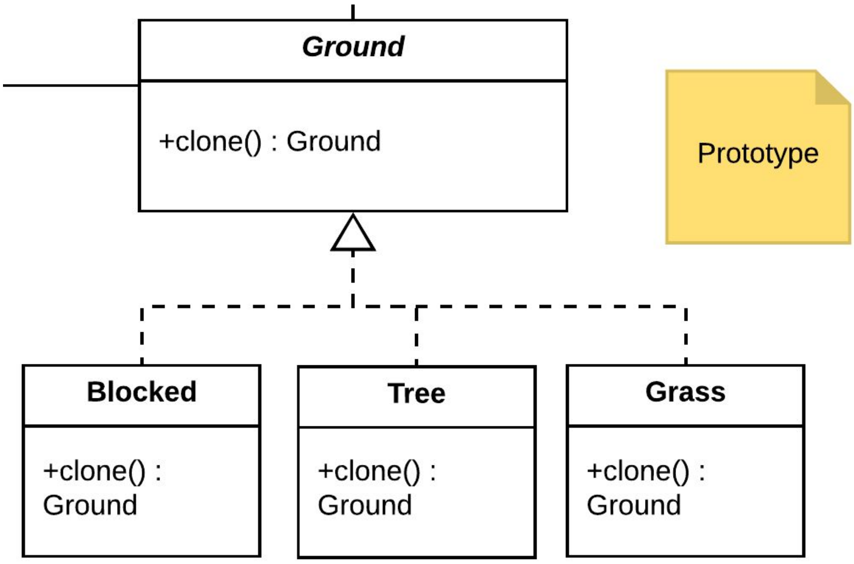
[Advantages]:

1. it isolates the piece classes.
2. it makes exchanging/ extending piece families easy.
3. it creates consistency among products.

[Implementation]:

The IPieceFactory defers creation of different teams to its concrete factory subclass (BanditPiece/ MerchantPiece), and the BanditPiece/ MerchantPiece factory creates specific piece having a particular implement. To create the different piece, the piece generator should use a different concrete factory.

Prototype Pattern:



[Motivation]:

Without creating new imageview to create a new background square we use the clone. Using prototype design pattern allows a GroundGenerator to create the customized ground without knowing their class or any details of how to create them.

[Advantages]:

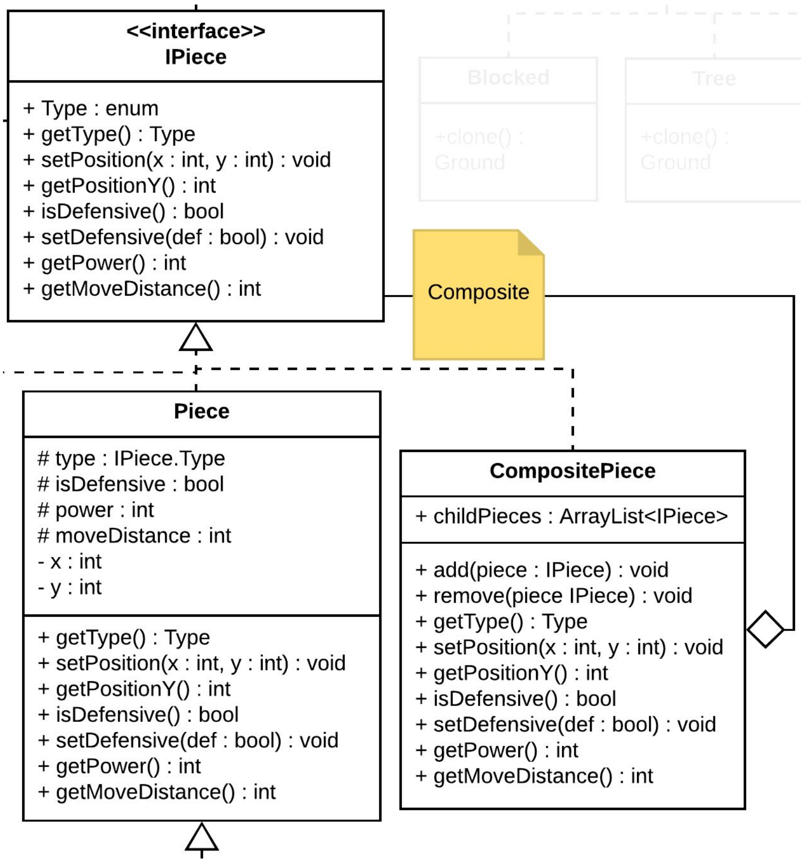
1. It creates the deep copy of the ground hierarchy.
2. it reduces the load of initialization of the specific ground.
3. It is easy to copy object recursively by calling the “clone” method on every member; it makes the program structure easier to understand and maintain.
4. it is good for reusability.

[Implementation]:

Let the superclass ground implements the Cloneable interface to indicate the clone method to make a field for field copy of the instance of that class

Structural Category:

Composite Pattern:



[Motivation]:

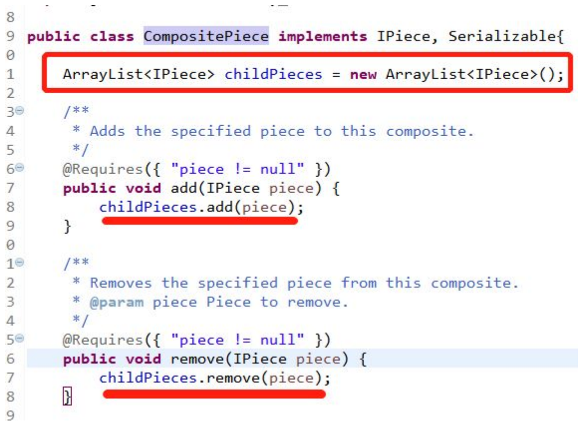
Using the composite design pattern it allows the board to be able to ignore the difference between individual and composed objects, the board will treat all of the pieces in the composite structure uniformly.

[Advantages]:

1. The new components can easily be added.
2. The board can treat composites and primitives uniformly.
3. The Primitive pieces can be recursively composed.

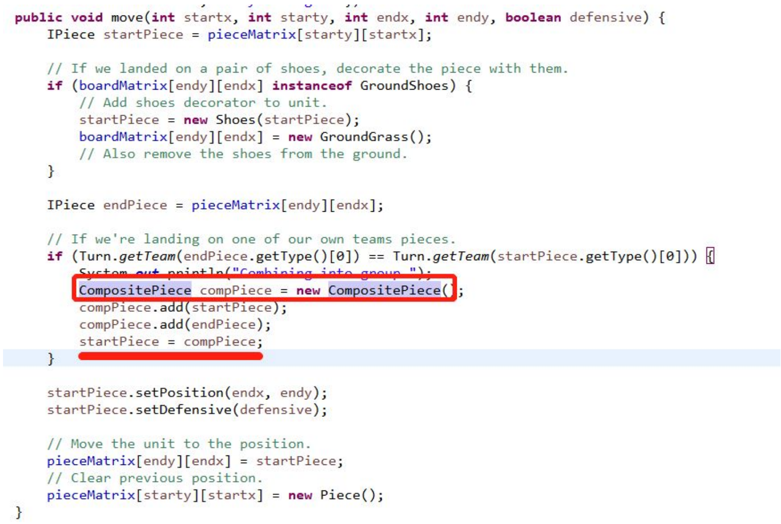
[Implementation]:

The composite Piece class maintains a collection of components. And also typically said, the composite methods are implemented by iterating that collection and invoking the appropriate method for each piece component in the group. Like following code :



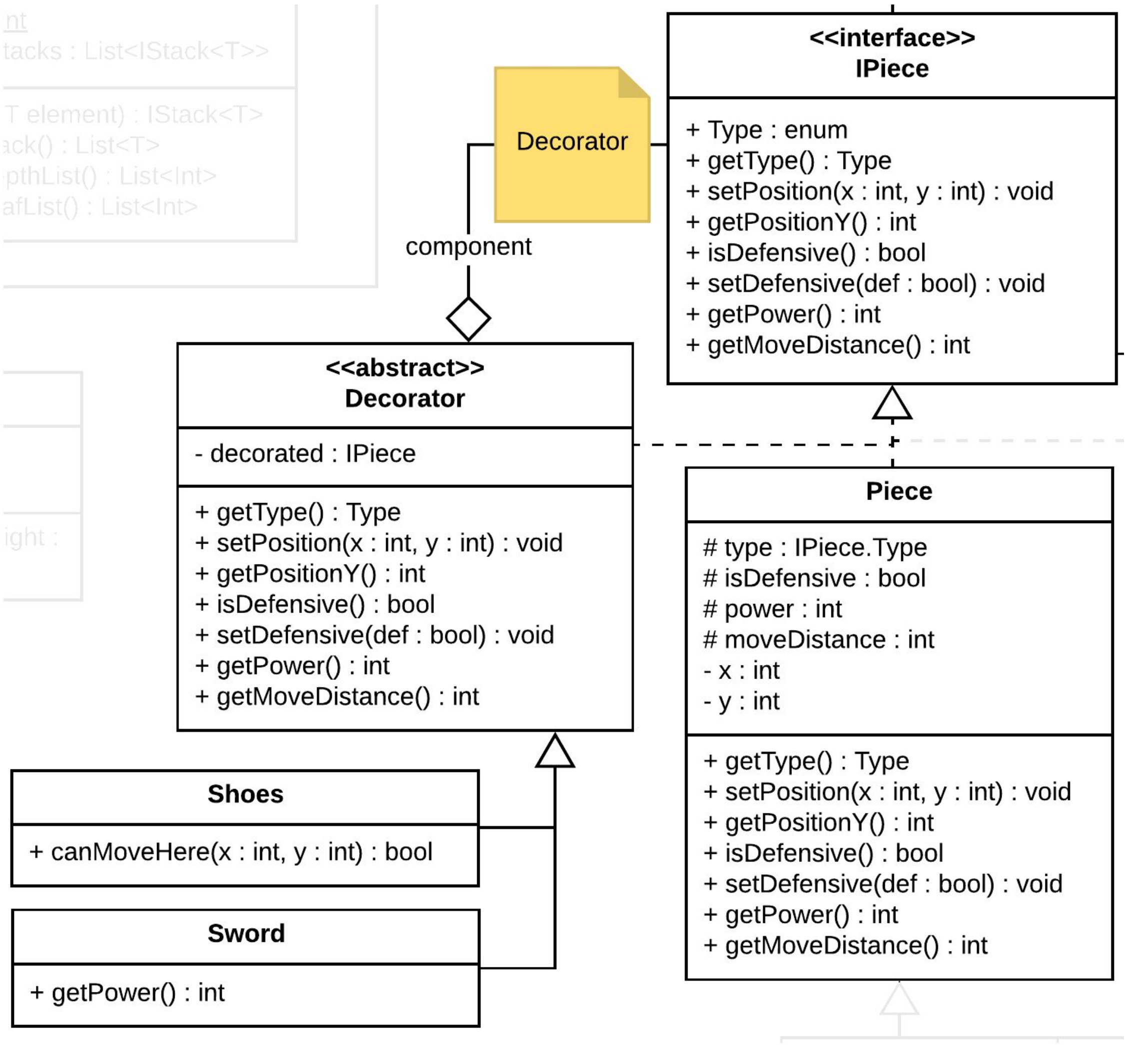
```
public class CompositePiece implements IPiece, Serializable{

    ArrayList<IPiece> childPieces = new ArrayList<IPiece>();

    /**
     * Adds the specified piece to this composite.
     */
    @Requires({ "piece != null" })
    public void add(IPiece piece) {
        childPieces.add(piece);
    }

    /**
     * Removes the specified piece from this composite.
     * @param piece Piece to remove.
     */
    @Requires({ "piece != null" })
    public void remove(IPiece piece) {
        childPieces.remove(piece);
    }
```

The following code let the board coordinate piece ignored the difference between the individual piece and composed piece



```
public void move(int startx, int starty, int endx, int endy, boolean defensive) {
    IPiece startPiece = pieceMatrix[starty][startx];

    // If we landed on a pair of shoes, decorate the piece with them.
    if (boardMatrix[endy][endx] instanceof GroundShoes) {
        // Add shoes decorator to unit.
        startPiece = new Shoes(startPiece);
        boardMatrix[endy][endx] = new GroundGrass();
        // Also remove the shoes from the ground.
    }

    IPiece endPiece = pieceMatrix[endy][endx];

    // If we're landing on one of our own teams pieces.
    if (Turn.getTeam(endPiece.getType()[0]) == Turn.getTeam(startPiece.getType()[0])) {
        System.out.println("Combining into group.");
        CompositePiece compPiece = new CompositePiece();
        compPiece.add(startPiece);
        compPiece.add(endPiece);
        startPiece = compPiece;
    }

    startPiece.setPosition(endx, endy);
    startPiece.setDefensive(defensive);

    // Move the unit to the position.
    pieceMatrix[endy][endx] = startPiece;
    // Clear previous position.
    pieceMatrix[starty][startx] = new Piece();
}
```

Decorator Pattern:



[Motivation]:

The game allows the user to choose between multiple game rules.In such a case decorator is implemented in order to avoid creating subclasses which becomes quite complex.The decorator pattern allows to extend or modify the behaviour of instance at runtime.

[Advantages]:

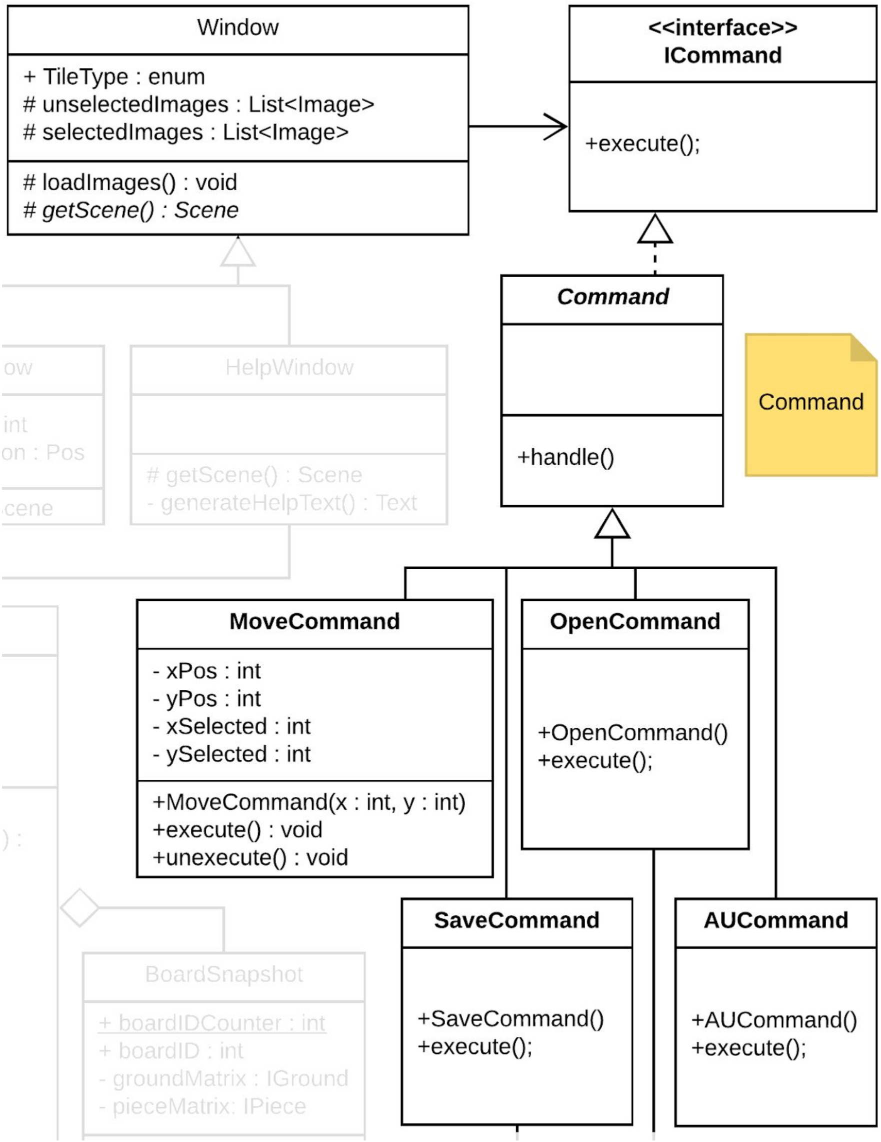
1. Add responsibilities to individual objects dynamically and transparently.
2. Useful when there are chances of combinatorial explosion when subclassing alone is used

[Implementation]:

The IPiece Interface created acts as a blueprint for the decorator class.The interface is implemented with the basic functionalities The decorator abstract class is the base decorator class from which many concrete decorator classes can be inherited.

Behavioral Category:

Command Pattern:



[Motivation]:

The command design pattern encapsulates command (handle method) in concrete command object allowing us to issue requests without knowing the requested operation or requesting object. Command design pattern provides the options to queue commands, undo action .

[Advantages]:

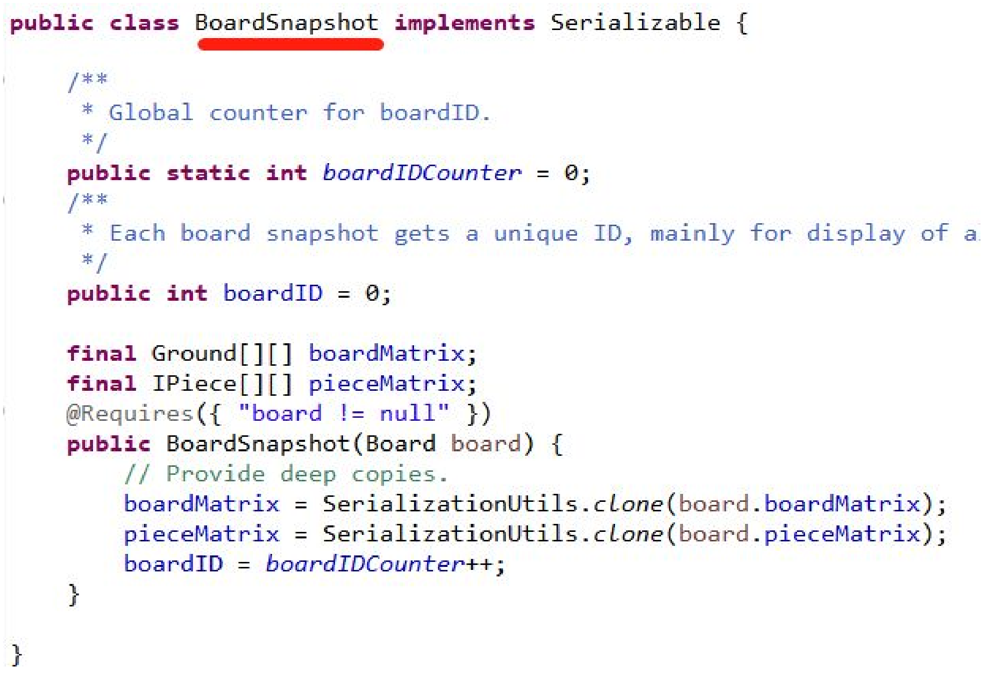
1. Extensions to add new commands is made easy without having to change existing code.

2. It decouples the classes that invoke the command from the object that knows how to execute the operation.
3. It encapsulate a request in an object.

[Implementation]:

As mentioned above, the command design pattern includes a part for undo of actions. There have two options in the game :

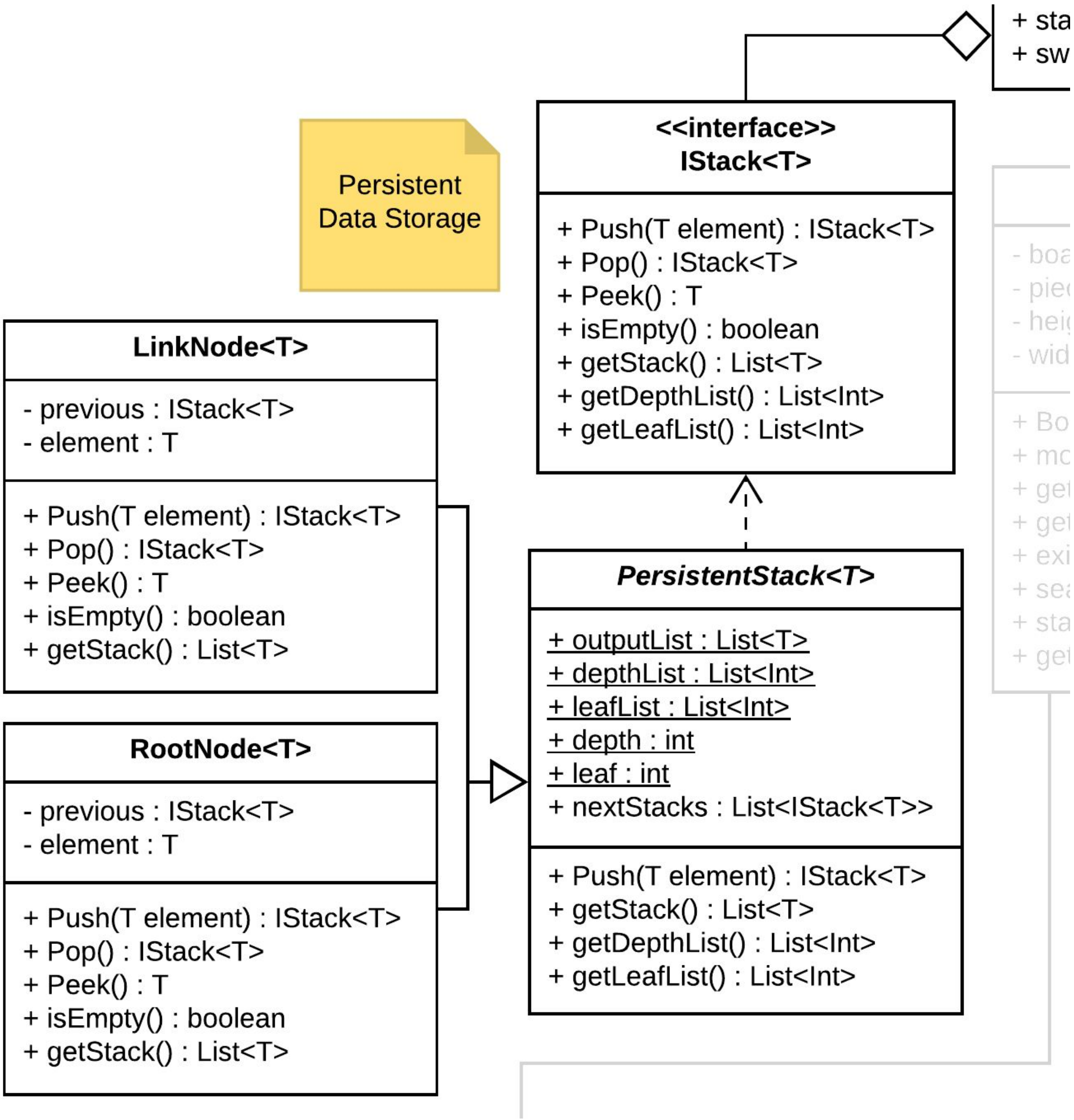
1. If the command performs reversing action, we can support undo by giving the command two stored function, one for acting, one for undoing it
2. But in this game, pieces can disappear after being captured, so we store a snapshot of the pre-execution state of the board.



```
public class BoardSnapshot implements Serializable {

    /**
     * Global counter for boardID.
     */
    public static int boardIDCounter = 0;
    /**
     * Each board snapshot gets a unique ID, mainly for display of a
     */
    public int boardID = 0;

    final Ground[][] boardMatrix;
    final IPiece[][] pieceMatrix;
    @Requires({ "board != null" })
    public BoardSnapshot(Board board) {
        // Provide deep copies.
        boardMatrix = SerializationUtils.clone(board.boardMatrix);
        pieceMatrix = SerializationUtils.clone(board.pieceMatrix);
        boardID = boardIDCounter++;
    }

}
```

The Command Design Pattern is implemented using the command interface which holds the execute method which asks the receiver to carry out the operation.The invoker holds the command and execute it using the execute method.Here the command class is the concrete class that binds the action and the receiver.

Persistent Data Structure:



[Motivation]:

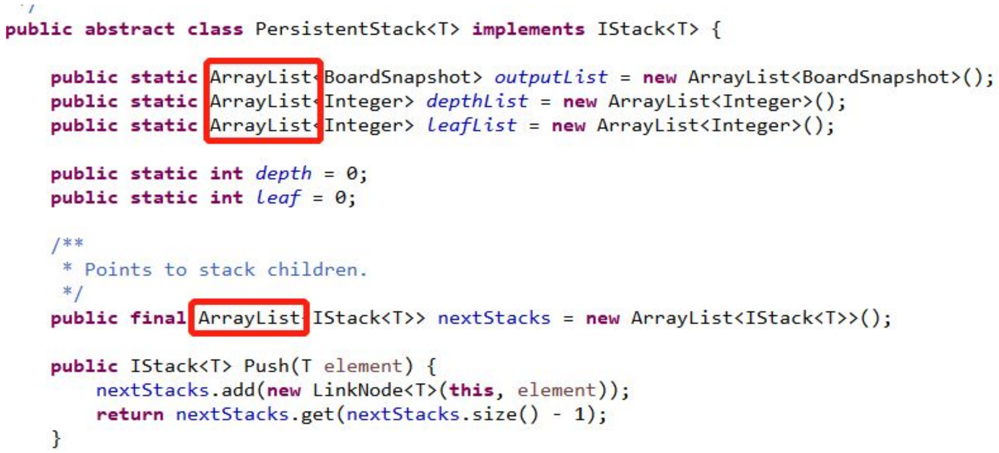
Compare with the partial persistent and the full persistent , the partial persistent past versions of the data structure may be queried, but only latest version may be modified, but in the case of full persistence, updates can be done on past versions in a branching model of time (i.e. each modification involves only one past version and creates a new version)

[Advantages]:

1. Using the persistent data structure makes the data immutable quite simple.
2. Don't need to lock the stack , which extremely improves concurrency.
3. And immutable objects are simpler to construct, test and use.

[Implementation]:

The easy to implement the persistent data structure is using the stack. Simply create a persistent singly Arraylist and limit insertion and deletion of the head of the list.



```
 */
public abstract class PersistentStack<T> implements IStack<T> {

    public static ArrayList<BoardSnapshot> outputList = new ArrayList<BoardSnapshot>();
    public static ArrayList<Integer> depthList = new ArrayList<Integer>();
    public static ArrayList<Integer> leafList = new ArrayList<Integer>();

    public static int depth = 0;
    public static int leaf = 0;

    /**
     * Points to stack children.
     */
    public final ArrayList<IStack<T>> nextStacks = new ArrayList<IStack<T>>();

    public IStack<T> Push(T element) {
        nextStacks.add(new LinkNode<T>(this, element));
        return nextStacks.get(nextStacks.size() - 1);
    }
```

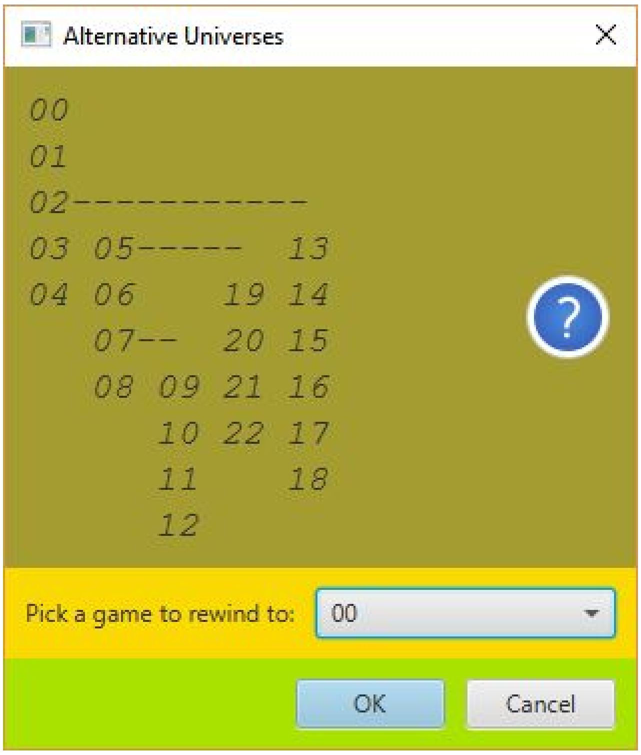
Since this class is persistent, popping a stack returns a new version of stack with the next item in the old stack as the new top.

[Game Rules]:

Each piece can toggle between attacking and defensive positions, the default being attacked. Attackive positions use the default movement and power of the pieces which favours speed over power. Defensive positions reduce piece speed by half but increases their power by 1. Attacking and defensive positions are set before the move is made. Once the piece has moved, the mode selected by the player will be active on the piece.

Alternative Universes

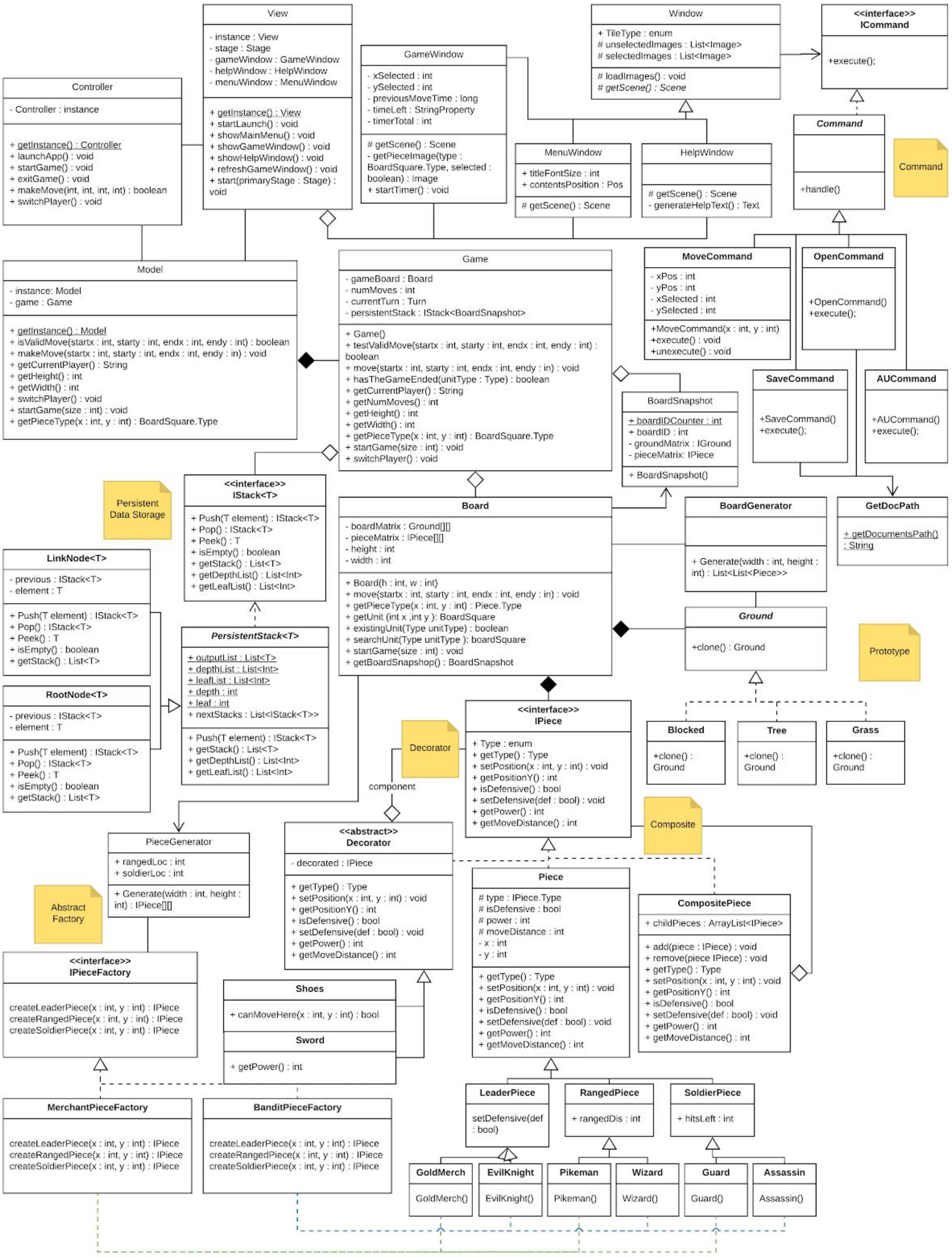
This shows the tree structure of game history where users can fork the tree at any given game. In this example, game node 02 has children 03, 05 and 13, game node 05 has children 06 and 19.



```
00
01
02-----------
03 05-----   13
04 06    19 14
   07--  20 15
   08 09 21 16
      10 22 17
      11    18
      12
```

Pick a game to rewind to: 00

Entire Class Diagram


View
- instance : View
- stage : Stage
- gameWindow : GameWindow
- helpWindow : HelpWindow
- menuWindow : MenuWindow
+ getInstance() : View
+ startLaunch() : void
+ showMainMenu() : void
+ showGameWindow() : void
+ showHelpWindow() : void
+ refreshGameWindow() : void
+ start(primaryStage : Stage) : void
GameWindow
- xSelected : int
- ySelected : int
- previousMoveTime : long
- timeLeft : StringProperty
- timerTotal : int
# getScene() : Scene
- getPieceImage(type : BoardSquare.Type, selected : boolean) : Image
+ startTimer() : void
Window
+ TileType : enum
# unselectedImages : List<Image>
# selectedImages : List<Image>
# loadImages() : void
# getScene() : Scene
<<interface>>
ICommand
+execute();
Controller
- Controller : instance
+ getInstance() : Controller
+ launchApp() : void
+ startGame() : void
+ exitGame() : void
+ makeMove(int, int, int, int) : boolean
+ switchPlayer() : void
MenuWindow
+ titleFontSize : int
+ contentsPosition : Pos
# getScene() : Scene
HelpWindow
# getScene() : Scene
- generateHelpText() : Text
Command
+handle()
Command
Model
- instance: Model
- game : Game
+ getInstance() : Model
+ isValidMove(startx : int, starty : int, endx : int, endy : int) : boolean
+ makeMove(startx : int, starty : int, endx : int, endy : in) : void
+ getCurrentPlayer() : String
+ getHeight() : int
+ getWidth() : int
+ switchPlayer() : void
+ startGame(size : int) : void
+ getPieceType(x : int, y : int) : BoardSquare.Type
Game
- gameBoard : Board
- numMoves : int
- currentTurn : Turn
- persistentStack : IStack<BoardSnapshot>
+ Game()
+ testValidMove(startx : int, starty : int, endx : int, endy : int) : boolean
+ move(startx : int, starty : int, endx : int, endy : in) : void
+ hasTheGameEnded(unitType : Type) : boolean
+ getCurrentPlayer() : String
+ getNumMoves() : int
+ getHeight() : int
+ getWidth() : int
+ getPieceType(x : int, y : int) : BoardSquare.Type
+ startGame(size : int) : void
+ switchPlayer() : void
MoveCommand
- xPos : int
- yPos : int
- xSelected : int
- ySelected : int
+MoveCommand(x : int, y : int)
+execute() : void
+unexecute() : void
OpenCommand
+OpenCommand()
+execute();
SaveCommand
+SaveCommand()
+execute();
AUCommand
+AUCommand()
+execute();
BoardSnapshot
+ boardIDCounter : int
+ boardID : int
- groundMatrix : IGround
- pieceMatrix: IPiece
+ BoardSnapshot()
Persistent Data Storage
<<interface>>
IStack<T>
+ Push(T element) : IStack<T>
+ Pop() : IStack<T>
+ Peek() : T
+ isEmpty() : boolean
+ getStack() : List<T>
+ getDepthList() : List<Int>
+ getLeafList() : List<Int>
Board
- boardMatrix : Ground[][]
- pieceMatrix : IPiece[][]
- height : int
- width : int
+ Board(h : int, w : int)
+ move(startx : int, starty : int, endx : int, endy : in) : void
+ getPieceType(x : int, y : int) : Piece.Type
+ getUnit (int x ,int y): BoardSquare
+ existingUnit(Type unitType) : boolean
+ searchUnit(Type unitType): boardSquare
+ startGame(size : int) : void
+ getBoardSnapshop() : BoardSnapshot
BoardGenerator
+ Generate(width : int, height : int) : List<List<Piece>>
GetDocPath
+ getDocumentsPath() : String
LinkNode<T>
- previous : IStack<T>
- element : T
+ Push(T element) : IStack<T>
+ Pop() : IStack<T>
+ Peek() : T
+ isEmpty() : boolean
+ getStack() : List<T>
PersistentStack<T>
+ outputList : List<T>
+ depthList : List<Int>
+ leafList : List<Int>
+ depth : int
+ leaf : int
+ nextStacks : List<IStack<T>>
+ Push(T element) : IStack<T>
+ getStack() : List<T>
+ getDepthList() : List<Int>
+ getLeafList() : List<Int>
Ground
+clone() : Ground
Prototype
RootNode<T>
- previous : IStack<T>
- element : T
+ Push(T element) : IStack<T>
+ Pop() : IStack<T>
+ Peek() : T
+ isEmpty() : boolean
+ getStack() : List<T>
<<interface>>
IPiece
+ Type : enum
+ getType() : Type
+ setPosition(x : int, y : int) : void
+ getPositionY() : int
+ isDefensive() : bool
+ setDefensive(def : bool) : void
+ getPower() : int
+ getMoveDistance() : int
Decorator
component
Blocked
+clone() : Ground
Tree
+clone() : Ground
Grass
+clone() : Ground
Composite
PieceGenerator
+ rangedLoc : int
+ soldierLoc : int
+ Generate(width : int, height : int) : IPiece[][]
<<abstract>>
Decorator
- decorated : IPiece
+ getType() : Type
+ setPosition(x : int, y : int) : void
+ getPositionY() : int
+ isDefensive() : bool
+ setDefensive(def : bool) : void
+ getPower() : int
+ getMoveDistance() : int
Piece
# type : IPiece.Type
# isDefensive : bool
# power : int
# moveDistance : int
- x : int
- y : int
+ getType() : Type
+ setPosition(x : int, y : int) : void
+ getPositionY() : int
+ isDefensive() : bool
+ setDefensive(def : bool) : void
+ getPower() : int
+ getMoveDistance() : int
CompositePiece
+ childPieces : ArrayList<IPiece>
+ add(piece : IPiece) : void
+ remove(piece IPiece) : void
+ getType() : Type
+ setPosition(x : int, y : int) : void
+ getPositionY() : int
+ isDefensive() : bool
+ setDefensive(def : bool) : void
+ getPower() : int
+ getMoveDistance() : int
Abstract Factory
<<interface>>
IPieceFactory
createLeaderPiece(x : int, y : int) : IPiece
createRangedPiece(x : int, y : int) : IPiece
createSoldierPiece(x : int, y : int) : IPiece
Shoes
+ canMoveHere(x : int, y : int) : bool
Sword
+ getPower() : int
MerchantPieceFactory
createLeaderPiece(x : int, y : int) : IPiece
createRangedPiece(x : int, y : int) : IPiece
createSoldierPiece(x : int, y : int) : IPiece
BanditPieceFactory
createLeaderPiece(x : int, y : int) : IPiece
createRangedPiece(x : int, y : int) : IPiece
createSoldierPiece(x : int, y : int) : IPiece
LeaderPiece
setDefensive(def : bool)
RangedPiece
+ rangedDis : int
SoldierPiece
+ hitsLeft : int
GoldMerch
GoldMerch()
EvilKnight
EvilKnight()
Pikeman
Pikeman()
Wizard
Wizard()
Guard
Guard()
Assassin
Assassin()